Design patterns

- A design pattern is a way of reusing abstract knowledge about a problem and its solution
  - Patterns are devices that allow programs to share knowledge about their design
- A pattern is a description of the problem and the essence of its solution
  - Documenting patterns is one way to reuse and share the information about how it is best to solve a specific design problem
- A pattern should be sufficiently abstract to be reused in different settings
- Patterns often rely on object characteristics such as inheritance and polymorphism
History of Design Patterns

- Architect Christopher Alexander
  - *A Pattern Language* (1977)
  - *A Timeless Way of Building* (1979)
- “Gang of four”
  - Erich Gamma
  - Richard Helm
  - Ralph Johnson
  - John Vlissides
  - *Design Patterns: Elements of Reusable Object-Oriented Software* (1995)
- Many since
- Conferences, symposia, books

Pattern elements

- Name
  - A meaningful pattern identifier
- Problem description
- Solution description
  - Not a concrete design but a template for a design solution that can be instantiated in different ways
- Consequences
  - The results and trade-offs of applying the pattern
Abstraction Levels

Patterns exist on different abstraction levels:

- **basic building blocks** (algorithms and components), provided by language or by libraries
  - e.g. hash tables, linked lists, sort algorithms, math
  - e.g. inheritance and polymorphism, encapsulation

- **design patterns**: general design problems in particular context concerning **several classes**
  - e.g. GOF-design patterns

- **architecture patterns**: architecture decisions concerning the **whole system** (or subsystem)
  - e.g. client-server, data centric etc.

---

Parallelization Strategy

**Finding Concurrency**

- Structure the problem to expose exploitable concurrency

**Algorithm Structure**

- Structure the algorithm to take advantage of concurrency

**Supporting Structure**

- Intermediate stage between Algorithm Structure and Implementation
  - program structuring
  - definition of shared data structures

**Implementation Mechanism**

- Mapping of the higher level patterns onto a programming environment
Finding concurrency - Overview

Overview (II)

- Is the problem large enough to justify the efforts for parallelizing it?
- Are the key features of the problem and the data elements within the problem well understood?
- Which parts of the problem are most computationally intensive?
Task decomposition

- How can a problem be decomposed into tasks that can execute concurrently?
- Goal: a collection of (nearly) independent tasks
  - Initially, try to find as many as tasks as possible (can be merged later to form larger tasks)
- A task can correspond to
  - A function call
  - Distinct iterations of loops within the algorithm (loop splitting)
  - Any independent sections in the source code

Task decomposition - goals and constraints

- Flexibility: the design should be flexible enough to be able to handle any numbers of processes
  - E.g. the number and the size of each task should be a parameter for the task decomposition
- Efficiency: each task should include enough work to compensate for the overhead of “generating” tasks and managing their dependencies
- Simplicity: tasks should be defined such that it keeps debugging and maintenance easy
Task decomposition

- Two tasks A and B are considered independent if

\[ I_A \cap O_B = \emptyset \]
\[ I_B \cap O_A = \emptyset \]
\[ O_A \cap O_B = \emptyset \]

---

Task decomposition - example

Solving a system of linear equations using the Conjugant Gradient Method

Given \( x_0, A, b \)
\[ p_0 = r_0 := b - Ax_0 \]
For \( i = 0, 1, 2, ... \)
\[ \alpha = r_i^T p_i \]
\[ \beta = (Ap_i)^T p_i \]
\[ \lambda = \alpha / \beta \]
\[ x_{i+1} = x_i + \lambda p_i \]
\[ r_{i+1} = r_i - \lambda Ap_i \]
\[ p_{i+1} = r_{i+1} - ((Ar_{i+1})^T p_i) / \beta \]

Can be executed in parallel
Data decomposition

- How can a problem’s data be decomposed into units that can be operated (relatively) independently?
  - Most computationally intensive parts of a problem are dealing with large data structures

- Common mechanisms
  - Array-based decomposition (e.g. see example on the next slides)
  - Recursive data structures (e.g. decomposing the parallel update of a large tree data structure)

Data decomposition - goals and constraints

- Flexibility: size and number of data chunks should be flexible (granularity)

- Efficiency:
  - Data chunks have to be large enough that the amount of work with the data chunk compensates for managing dependencies
  - Load balancing

- Simplicity:
  - Complex data types difficult to debug
  - Mapping of local indexes to global indexed often required
Grouping tasks

- How can (small) tasks be grouped to simplify the management of dependencies?
- Tasks derived from task decomposition or data decomposition do not necessarily constitute a flat tree
  - Might be derived from the same high level operation
  - Might have similar constraints

Ordering tasks

- Tasks might be truly independent (trivial parallelism)
  - E.g. search operations
  - E.g. Monte-Carlo Simulations
- Tasks must run simultaneously
  - E.g. often occurring in data decomposition
- Temporal dependency: constraint on the order in which a collection of tasks execute
  - E.g. Task B needs the result of Task A
Temporal dependency - example
Solving a system of linear equations using the Conjugate Gradient Method

Given the vectors \( \mathbf{x}_0, \mathbf{A}, \mathbf{b} \)

\[
\mathbf{p}_0 = \mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0
\]

For \( i = 0, 1, 2, \ldots \)

\[
\begin{align*}
\alpha &= \mathbf{r}_i^T \mathbf{p}_i \\
\beta &= (\mathbf{A}\mathbf{p}_i)^T \mathbf{p}_i \\
\lambda &= \frac{\alpha}{\beta} \\
\mathbf{x}_{i+1} &= \mathbf{x}_i + \lambda \mathbf{p}_i \\
\mathbf{r}_{i+1} &= \mathbf{r}_i - \lambda \mathbf{A}\mathbf{p}_i \\
\mathbf{p}_{i+1} &= \mathbf{r}_{i+1} - ((\mathbf{A}\mathbf{r}_{i+1})^T \mathbf{p}_i) / \beta
\end{align*}
\]

Data sharing

- Up to this point we have
  - A collection of tasks that can execute independently
  - A data decomposition corresponding to the collection of concurrent tasks
  - Dependencies among the tasks
- Question: how is data shared amongst the tasks?
  - Shared variables (e.g. in OpenMP)
  - Message passing, e.g. ghost cell update
Finding concurrency

- **Result**
  - A task decomposition that identifies tasks that can execute concurrently
  - A data decomposition that identifies data local to each task
  - A way of grouping tasks and ordering them according to temporal constraints

Algorithm structure

- **Organize by tasks**
  - Task Parallelism
  - Divide and Conquer

- **Organize by data decomposition**
  - Geometric decomposition
  - Recursive data

- **Organize by flow of data**
  - Pipeline
  - Event-based coordination
Task parallelism (I)

- Problem can be decomposed into a collection of tasks that can execute concurrently
- Tasks can be completely independent (embarrassingly parallel) or can have dependencies among them
- All tasks might be known at the beginning or might be generated dynamically

Task parallelism (II)

- Tasks:
  - There should be at least as many tasks as UEs (typically many, many more)
  - Computation associated with each task should be large enough to offset the overhead associated with managing tasks and handling dependencies
- Dependencies:
  - Ordering constraints: sequential composition of task-parallel computations
  - Shared-data dependencies: several tasks have to access the same data structure
Task scheduling

• Schedule: the way in which tasks are assigned to UEs for execution
• Goal: load balance - minimize the overall execution of all tasks
• Two classes of schedule:
  - Static schedule: distribution of tasks to UEs is determined at the start of the computation and not changed anymore
  - Dynamic schedule: the distribution of tasks to UEs changes as the computation proceeds
Static schedule

- Tasks are associated into blocks
- Blocks are assigned to UEs
- Each UE should take approximately same amount of time to complete task
- Static schedule usually used when
  - Availability of computational resources is predictable (e.g. dedicated usage of nodes)
  - UEs are identical (e.g. homogeneous parallel computer)
  - Size of each task is nearly identical

Dynamic scheduling

- Used when
  - Effort associated with each task varies widely/is unpredictable
  - Capabilities of UEs vary widely (heterogeneous parallel machine)
- Common implementations:
  - usage of task queues: if a UE finishes current task, it removes the next task from the task-queue
  - Work-stealing:
    - each UE has its own work queue
    - once its queue is empty, a UE steals work from the task queue of another UE
Dynamic scheduling

- **Trade-offs:**
  - Fine grained (=shorter, smaller) tasks allow for better load balance
  - Fine grained task have higher costs for task management and dependency management

---

Divide and Conquer algorithms
Divide and Conquer

• A problem is split into a number of smaller sub-problems
• Each sub-problem is solved independently
• Sub-solutions of each sub-problem will be merged to the solution of the final problem
• Problems of Divide and Conquer for Parallel Computing:
  - Amount of exploitable concurrency decreases over the lifetime
  - Trivial parallel implementation: each function call to `solve` is a task on its own. For small problems, no new task should be generated, but the `baseSolve` should be applied

Divide and Conquer

• Implementation:
  - On shared memory machines, a divide and conquer algorithm can easily be mapped to a `fork/join` model
    • A new task is forked (=created)
    • After this task is done, it joins the original task (=destroyed)
  - On distributed memory machines: task queues
    • Often implemented using the Master/Worker framework - discussed later
Divide and Conquer

```c
int solve ( Problem P )
{
    int solution;

    /* Check whether we can further partition the problem */
    if (baseCase(P) ) {
        solution = baseSolve(P); /* No, we can’t */
    } else { /* yes, we can */
        Problem subproblems[N];
        int subsolutions[N];

        subproblems = split (P); /* Partition the problem */
        for ( i=0; i < N; i++ ) {
            subsolutions[i] = solve ( subproblems[i]);
        }
        solution = merge (subsolutions);
    }
    return ( solution );
}
```

Task Parallelism using Master-Worker framework

Master Process

Worker Process 1

Worker Process 2

Result queue

Task queue
Task Parallelism using work stealing

Worker Process 1

Worker Process 2

Geometric decomposition

- For all applications relying on data decomposition
  - All processes should apply the same operations on different data items
- Key elements:
  - Data decomposition
  - Exchange and update operation
  - Data distribution and task scheduling
2-D Example Laplace equation

- Parallel domain decomposition
  - Data exchange at process boundaries required
- Halo cells / Ghost cells
  - Copy of the last row/column of data from the neighbor process

Recursive Data

- Typically applied in recursive data structures
  - Lists, trees, graphs
- Data decomposition: recursive data structure is completely decomposed into individual elements
- Example: prefix scan operation
  - Each process has an element of an overall structure such as a linked list, e.g. an integer \( x \)
    - Lets denote the value of the \( x \) on process \( i \) as \( x_i \)
  - At the end of the prefix scan operation process \( k \) holds the sum of all elements of \( x_i \) for \( i=0...k \)
Recursive data (II)

- Example for eight processes

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Before prefix scan</td>
<td>(x_0)</td>
<td>(x_1)</td>
<td>(x_2)</td>
<td>(x_3)</td>
<td>(x_4)</td>
<td>(x_5)</td>
<td>(x_6)</td>
</tr>
<tr>
<td>After prefix scan</td>
<td>(x_0)</td>
<td>(x_0 + x_1)</td>
<td>(x_0 + x_1 + x_2)</td>
<td>(x_0 + x_1 + x_2 + x_3)</td>
<td>(x_0 + x_1 + x_2 + x_3 + x_4)</td>
<td>(x_0 + x_1 + x_2 + x_3 + x_4 + x_5)</td>
<td>...</td>
</tr>
</tbody>
</table>

Sequential implementation

Each process forwards its sum to the next process
- \(n\) messages/time steps required for \(n\) processes

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_0)</td>
<td>(x_1)</td>
<td>(x_2)</td>
<td>(x_3)</td>
<td>(x_4)</td>
<td>(x_5)</td>
<td>(x_6)</td>
<td>(x_7)</td>
</tr>
<tr>
<td>(x_0)</td>
<td>(x_0 + x_1)</td>
<td>(x_0 + x_1 + x_2)</td>
<td>(x_0 + x_1 + x_2 + x_3)</td>
<td>(x_0 + x_1 + x_2 + x_3 + x_4)</td>
<td>(x_0 + x_1 + x_2 + x_3 + x_4 + x_5)</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Recursive data approach

\[ \sum (x_0 : x_1) \sum (x_1 : x_2) \sum (x_2 : x_3) \sum (x_3 : x_4) \sum (x_4 : x_5) \sum (x_5 : x_6) \sum (x_6 : x_7) \]

Recursive Data (III)

- Very fine grained concurrency
- Restructuring of the original algorithm often required
- Parallel algorithm requires substantially more work, which can however be executed in less time-steps
Pipeline pattern

- Calculation can be viewed in terms of **data flowing** through a sequence of stages
- Computation performed on many data sets
  - Compare to pipelining in processors on the instruction level

\[
\begin{align*}
\text{Pipeline stage 1:} & \quad C_1 \quad C_2 \quad C_3 \quad C_4 \quad C_5 \quad C_6 \\
\text{Pipeline stage 2:} & \quad C_1 \quad C_2 \quad C_3 \quad C_4 \quad C_5 \quad C_6 \\
\text{Pipeline stage 3:} & \quad C_1 \quad C_2 \quad C_3 \quad C_4 \quad C_5 \quad C_6 \\
\text{Pipeline stage 4:} & \quad C_1 \quad C_2 \quad C_3 \quad C_4 \quad C_5 \quad C_6
\end{align*}
\]

Pipeline pattern (II)

- Amount of concurrency limited to the number of stages of the pipeline
- Patterns works best, if amount of work performed by various stages is roughly equal
- Filling the pipeline: some stages will be idle
- Draining the pipeline: some stages will be idle
- Non-linear pipeline: pattern allows for different execution for different data items
Pipeline pattern (III)

- Implementation:
  - Each stage typically assigned to a process/thread
  - A stage might be a data-parallel task itself
  - Computation per task has to be large enough to compensate for communication costs between the tasks

Event-based coordination pattern

- Pipeline pattern assumes a regular, non-changing data flow
- Event-based coordination assumes irregular interaction between tasks
- Real world example:
  - Data items might flow in both directions
  - Each data item might take a different path
- Major problem: deadlock avoidance
Event-based coordination (II)

- Examples:
  - Any discrete event simulations
    - E.g. traffic control systems
    - Simulation of digital circuits

Supporting structures (I)

- Algorithm structures focus on expressing concurrency on the algorithmic level
  - Problem-oriented
- Supporting structures describe software constructions for parallel algorithms
  - Intermediate stage between problem and implementation
Supporting structures (II)

Program Structures

- SPMD
- Master / Worker
- Loop Parallelism
- Fork / Join

Data Structures

- Shared Data
- Shared Queue
- Distributed Array

SPMD

- SPMD - Single Program Multiple Data
- Each UE carries out similar/identical operations
- Interaction between UEs performance critical
  - Basically all applications scaling up to several thousand nodes/processors are written in the SPMD style
SPMD (II)

• Basic elements:
  - Initialize: establish common context on each UE
  - Obtain unique identifier: e.g. using MPI_Comm_rank()
  - Run the same program on each UE using the unique identifier to differentiate behavior on different UEs
    • Differentiation could also be done based on data items
  - Distribute data: e.g. geometric decomposition
  - Finalize

Example: Numerical Integration

• Anti-differentiation: Given a function \( f(x) \), find a function \( F(x) \) with the property that \( F'(x) = f(x) \)
  • Example: \( f(x) = ax^n \) \( \rightarrow \) \( F(x) = \frac{1}{n+1} ax^{n+1} + c \)

• Calculating the Integral of a function \( \int_a^b f(x)dx = F(b) - F(a) \)

• Graphical interpretation

![Graphical representation of definite integral](image)
Numerical Integration (II)

- Some antiderivatives can not be determined easily, e.g.
  \[ F(x) = \int e^{x^2} \, dx = \sum_{k=0}^{n} \frac{x^{2k+1}}{(2k+1)k!} \]
- ...some can not be determined at all
  \[ F(x) = \int \frac{e^x}{x} \, dx = ? \]
  \[ \rightarrow \text{numerical approximations required} \]
- (Composite) trapezoid rule
  \[ \int_{a}^{b} f(x) \, dx = \sum_{i=1}^{n} \int_{x_{i-1}}^{x_{i}} f(x) \, dx \approx \frac{1}{2} \sum_{i=1}^{n} (x_{i} - x_{i-1})[f(x_{i-1}) + f(x_{i})] \]

Trapezoid Rule

- Graphical interpretation

![Graphical interpretation of the trapezoid rule](image)
Sequential algorithm

```c
#include <stdio.h>

int main ( int argc, char **argv )
{
    int i, num_steps=100000;
    double x, xn, pi, step, sum=0.0;

    /* Required input:
        - a,b : boundaries of the integral
        - f(x): function
    */

    step = (b-a)/num_steps;
    for (i=0; i<num_steps; i++) {
        x   = i * step;
        xn  = (i+1) * step;
        sum = sum + 0.5*(xn-x)*(f(x)+f(xn));
    }
    return (0);
}
```

1st MPI parallel algorithm

```c
... int rank, size, start, end, i, num_steps=100000;
    double x, xn, end, step, sum, lsum=0.0;

    MPI_Init ( &argc, &argv );
    MPI_Comm_rank (MPI_COMM_WORLD, &rank );
    MPI_Comm_size (MPI_COMM_WORLD, &size );

    step  = (b-a)/num_steps;
    start = rank * num_steps/size;
    end   = start + num_steps/size;

    for (i=start; i<end; i++) {
        x   = i * step;
        xn  = (i+1) * step;
        lsum = lsum + 0.5*(xn-x)*(f(x)+f(xn));
    }
    MPI_Allreduce (lsum,sum,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
    MPI_Finalize ();
...
1st MPI parallel algorithm (II)

- Comments:
  - Algorithm does not handle if `num_steps % size`

```c
start = rank * num_steps/size;
end   = start + num_steps/size;
rem   = num_steps % size;

if ( rem != 0 ){
    if ( rank < rem ) {
        start = start + rank;
        end   = end + rank + 1;
    }
    else {
        start = start + rem;
        end   = end + rank;
    }
}
```

Master-Worker pattern (II)

- Useful if
  - Workload associated with tasks are highly variable - MW has ‘built-in’ load balancing
  - Capabilities of PEs are strongly varying
  - Tasks are not tightly coupled - each worker process typically only has to communicate with the master process but not with other workers

- Not useful usually if the computationally intensive part of the program structure is organized in a big loop
Master-Worker pattern (III)

- Approach:
  - Two logically different entities: master process managing a work-queue, worker processes executing a task assigned to them by the master
  - Completion: explicit notification of master to worker processes typically required
    - Can become very complicated for adaptive and recursive problems, where a worker can also ‘generate’ new tasks

Master-Worker pattern

- Master/worker pattern works well, if a master has sufficient worker processes
- Master process can become a bottleneck if tasks are too small and number of worker processes is very large
Loop parallelism

- In many scientific applications, the most compute intensive part is organized in a large loop
- Splitting the loop execution onto different processes is a straightforward parallelization, if the internal structure (=dependencies) allow that
  - E.g. the numerical integration example on slides 11-17 is a loop parallel algorithm as well
- Most applications of the loop parallelism pattern rely on OpenMP

---

Numerical integration - OpenMP example

```c
#include <stdio.h>
#include "omp.h"

int main ( int argc, char **argv )
{
    int i, num_steps=100000;
    double x, xn, pi, step, sum=0.0;

    step = (b-a)/num_steps;

#pragma omp parallel for private(x,xn) reduction(+:sum)
    for ( i=0; i<num_steps; i++) {
        x = i * step;
        xn = (i+1) * step;
        sum = sum + 0.5*(xn-x)*(f(x)+f(xn));
    }
    return (0);
}
```
Fork/join pattern

- Useful, if the number of concurrent tasks varies during execution
  - Tasks are created dynamically (= forked)
  - Tasks are terminated when done (= join with parents)

Fork/join pattern (II)

- Can be useful for divide and conquer algorithms
- Often used with OpenMP
  - Can be used with MPI - 2 dynamic process management as well
- Creating and terminating processes/threads has a significant overhead